transalpyne: a language for automatic transposition

Luca de Feo, Éric Schost

`transalpyne`: a language for automatic transposition

Luca De Feo
LIX, École Polytechnique, France
luca.defeo@polytechnique.edu

Éric Schost
ORCCA, CSD, University of Western Ontario, Canada
eschost@uwo.ca

Abstract

We present here `transalpyne`, a scripting language, to be executed on top of a computer algebra system, that is specifically conceived for automatic transposition of linear functions. Its type system is able to automatically infer all the possible linear functions realized by a computer program. The key feature of `transalpyne` is its ability to transform a computer program computing a linear function in another computer program computing the transposed linear function. The time and space complexity of the resulting program are similar to the original ones.

Categories and Subject Descriptors I.1.3 [*Symbolic and Algebraic Manipulation*]: Languages and Systems—Special-purpose algebraic systems

General Terms Languages, Algorithms

Keywords Transposition principle, Tellegen's principle, linear algebra, programming languages

1. Introduction

Computer Algebra is devoted to developing algorithms to work on symbolic representations of mathematical objects. Linear maps over vector spaces or, more generally, free modules are often represented by matrices, either in dense or sparse form. The so-called black-box model gives another way of representing a linear application $L : V \to W$: a computer program that on any input $v \in V$ gives as output $L(v)$ is taken as a symbolic representation of L; this of course assumes a precise computer representation of the elements of V and W.

Since the seminal paper [25], computer algebraists have developed algorithms to work with black-box represented linear maps. In the black-box model, algorithms are only allowed to query the black-box by feeding an input to the black-box program and reading its output; no other information on the linear map can be obtained, in particular the source code of the program cannot be analyzed. The complexity of black-box algorithms is measured as in the computational model being used to describe the algorithm, plus the number of calls to the black-box program is taken into account as a special parameter.

In the black-box model, algorithms are known to compute the minimal polynomial, the determinant, the inverse, the rank [17, 25] and the characteristic polynomial [5, 6, 23]. This model is interesting whenever the matrix representing the linear map is too big to allow efficient processing by a computer program, however its information can easily be compressed in a black-box program: sparse or Vandermonde matrices are a classical example.

On the other side, Algebraic Complexity studies the complexity of computer programs that perform algebraic computations by abstracting from the actual representation of algebraic elements. Only arithmetic operations in the ring of interest are accounted for. One of the standard models used in algebraic complexity is the arithmetic circuit: a directed acyclic graph (DAG) is used to represent the flow of arithmetic evaluations, each node of the DAG accounts for one arithmetic operation (usually $+$ or $*$).

In particular, arithmetic circuits can be used to represent black-box programs computing linear maps. Then it is a well known theorem, [2, Th. 13.20] that a linear map and its transpose have similar algebraic complexities in the arithmetic circuit model; this is often known as *transposition theorem* or *Tellegen's theorem*. By a well known equivalence [2, Lemma 13.17], the transposition theorem extends to the straight-line program (SLP) model. These results justify the fact that some extensions of the black-box model allow black-box algorithms to query the linear form as well as its transpose.

Besides that, extensions of the transposition theorem to the Random Acces Machine (RAM) model have been successfully applied in computer algebra to develop efficient algorithms [1, 4, 12, 19, 22]. The key to all these results is to realize that a certain map is the transpose of some other well known linear map L. Then, an efficient algorithm in the RAM model for L is translated to the arithmetic circuit model, the transposition theorem is applied and the resulting arithmetic circuit is translated back to a RAM algorithm. All the papers use *ad hoc* transformations to/from the arithmetic circuit model but give no general technique to perform such translation; the only notable exception is [1] that defines a very restricted language –not far away from the SLP paradigm– in which a constructive proof of the transposition principle is possible.

Here we present an extension of [1] that allows to automatically treat a wider class of programs. We reserve the theoretical details of the construction for a forthcoming paper and focus instead on its implementation. We are currently developing an extension language for python, called `transalpyne`, for which transposition can be automatically performed by the compiler/interpreter.

One of the key features of `transalpyne` is the possibility to automatically discover all the possible *linearizations* of a program. In fact, many linear functions can correspond to the same computer program: in the case of multiplication of polynomials, for example, the same program corresponds to two linear functions, namely left and right multiplication by a constant. `transalpyne` uses an algorithm similar to the type inference of statically typed functional languages [3] to discover all of these linearizations.

For each discovered linearization, the compiler/interpreter generates the correct transposition. It can be shown that the algebraic complexity of the resulting program is similar to the one of the orig-

inal program. In the next sections we summarize the theory and the practice of transposition in `transalpyne`.

2. Arithmetic circuits

In this section we briefly present the arithmetic circuit model. For convenience, our presentation slightly deviates from textbooks; for a more classical and extensive treatment see [2, 24].

2.1 Basic definitions

DEFINITION 1 (Arithmetic operator, arity). *Let R be a ring. An arithmetic operator over R is a function $f : R^i \to R^o$ for some $i, o \in \mathbb{N}$. We set $R^0 = 0$, the zero R-module. Here i is called the* in-arity *of f or simply* arity*, o is called the* out-arity *of f.*

DEFINITION 2 (Arithmetic basis). *Let R be a ring. An arithmetic R-basis is a (not necessarily finite) set of arithmetic operators over R.*

The arithmetic basis we will work with is the *linear basis*, denoted by $\mathcal{L}$. It is composed of

$$\begin{array}{lll} + : R \times R \to R & *_a : R \to R & \mathrm{H} : R \to R \times R \\ a, b \mapsto a + b & b \mapsto ab & a \mapsto a, a \end{array} \qquad (\mathcal{L})$$

$$\begin{array}{ll} 0 : 0 \to R & \omega : R \to 0 \\ \bot \mapsto 0 & a \mapsto \bot \end{array}$$

where we denote by $\bot$ the unique element of 0 to avoid confusion with the 0 of R. Arithmetic circuits are directed acyclic multigraphs carrying information from an arithmetic basis; the formal definition follows.

DEFINITION 3 (Arithmetic node). *Let R be a ring and $\mathcal{B}$ be an R-basis. A node over $(R, \mathcal{B})$ is a tuple $v = (I, O, f)$ such that*

- *I and O are finite ordered sets,*
- *f is either an element of $\mathcal{B}$ or the special value $\emptyset$.*
- *If $f = \emptyset$, one of the two following conditions must hold:*
  - *I is a singleton and O is empty, in this case we say that v is an* input node*;*
  - *I is empty and O is a singleton, in this case we say that v is an* output node*.*
- *If $f \neq \emptyset$, the cardinality of I matches the in-arity of f and the cardinality of O matches the out-arity of f; in this case we say that v is an* evaluation node*.*

We call *input ports* the elements of I and *output ports* the elements of O, which we denote respectively by $\mathrm{in}(v)$ and $\mathrm{out}(v)$. The cardinalities of I and O are called, respectively, the *in-degree* and *out-degree* of v. We call f the *value* of v and write $\beta(v)$ for it.

DEFINITION 4 (Arithmetic circuit). *Let R be a ring and $\mathcal{B}$ be an R-basis. An arithmetic circuit over $(R, \mathcal{B})$ is a tuple $C = (V, E, \leqslant, \leqslant_i, \leqslant_o)$ such that*

1. *V is a finite set of nodes over $(R, \mathcal{B})$;*
2. *$<$ is a total order on V, $<_i$ is a total order on the input nodes in V, $<_o$ is a total order on the output nodes in V;*
3. *let $I = \biguplus_{v \in V} \mathrm{in}(v)$ and $O = \biguplus_{v \in V} \mathrm{out}(v)$, then E is a bijection from O to I such that $E(o) = i$ implies that $o \in \mathrm{out}(v)$, $i \in \mathrm{in}(v')$ and $v \lneq v'$.*

It is useful to see E as a set of pairs (o, i) with $i \in I$ and $o \in O$. Then the elements of E are called the *edges* of the circuit. The edges *incident* to $v \in V$ are all the $(o, i) \in E$ such that $i \in \mathrm{in}(v)$; the edges *stemming* from $v \in V$ are all the $(i, o) \in E$ such that $o \in \mathrm{out}(v)$. An edge stemming from v and incident to v' is said to *connect* v to v'. We call *inputs* and *outputs* of a circuit, respectively, the input and output nodes in V; which we denote by $\mathrm{in}(C)$ and $\mathrm{out}(C)$.

DEFINITION 5 (Size, depth). *Let C be a circuit over $(R, \mathcal{B})$. The size of C, denoted by $\mathrm{size}(C)$ is the number of evaluation nodes in V; the depth of C, denoted by $\mathrm{depth}(C)$ is the length of the longest directed path –in a graph-theoretic sense– in (V, E).*

Sometimes it is useful to only count certain nodes. Let $X \subset \mathcal{B}$, the X-weighted size of C, denoted by $\mathrm{size}_X(C)$ is the number of nodes $v \in V$ such that $\beta(v) \in X$.

Figure 1 shows an example of arithmetic circuit, the analogy with multiDAGs is evident. We draw input and output nodes in square boxes and evaluation nodes in round boxes.

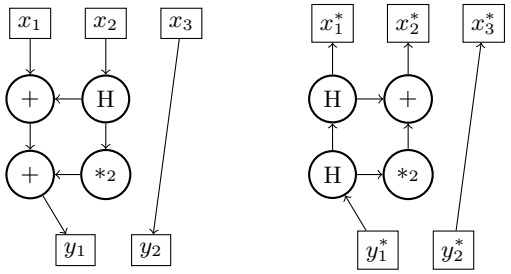

Figure 1. Two arithmetic circuits over $\mathcal{L}$. The linear map $y_1 = x_1 + 3x_2, y_2 = x_3$ is computed by the circuit on the left and its transpose is computed by the circuit on the right.

Circuits are endowed with the usual semantics consisting in the evaluation of the arithmetic operations on their inputs. We denote by eval_C the function $R^i \to R^o$ computed by the circuit C. For a circuit over $(R, \mathcal{L})$ it can be shown that eval_C is a linear operator; we skip the formal definitions and proofs for conciseness.

2.2 The transposition theorem

For a circuit over the basis $\mathcal{L}$, each node can be viewed as a linear operator and the arrows can be understood as composing operators in a suitable way to obtain eval_C. By reversing the flow and transposing the operator computed at each node, one obtains a circuit that computes the transposed operator.

DEFINITION 6. *Dual circuit Let $C = (V, E, \leqslant, \leqslant_I, \leqslant_O)$ be a circuit over $(R, \mathcal{L})$, the dual circuit of C, denoted by C^*, is the arithmetic circuit*

$$C^* = (V^*, E^{-1}, \leqslant', \leqslant'_i, \leqslant'_o)$$

where for any node $v = (I, O, f)$ in V there is a node $v^* = (O, I, f^*)$ in V^* where

$$f^* = \begin{cases} f & \text{if } f = *_a, \\ + & \text{if } f = \mathrm{H}, \\ \mathrm{H} & \text{if } f = +, \\ \omega & \text{if } f = 0, \\ 0 & \text{if } f = \omega; \end{cases} \qquad (1)$$

and for any input node $v = (\emptyset, O, \emptyset)$ there is an output node $v^* = (O, \emptyset, \emptyset)$ and vice versa*.*

The orderings $\leqslant'$, $\leqslant'_i$ and $\leqslant'_o$ are defined as follows:

$$v \leqslant v' \;\Leftrightarrow\; v'^* \leqslant' v^*, \qquad (2)$$

$$v \leqslant_o v' \;\Leftrightarrow\; v^* \leqslant'_o v'^*, \qquad (3)$$

$$v \leqslant_i v' \;\Leftrightarrow\; v^* \leqslant'_i v'^*. \qquad (4)$$

In particular, this makes (V', E^{-1}) the reverse graph of (V, E) in a graph-theoretic sense. Figure 1 shows two circuits that each other's dual. We now state the transposition theorem, for a proof see [2, Th. 13.20].

THEOREM 1 (Transposition theorem). *Let C be a circuit over $(R, \mathcal{L})$ that computes a linear application f, then C^* computes the transposed linear application f^*.*

COROLLARY 1. *A linear function $f : R^n \to R^m$ and its transpose can be computed by arithmetic circuits of same sizes and depths. In particular if C computes f and C^* computes f^*,*

$$\text{size}_{\{+\}}(C) = \text{size}_{\{\text{H}\}}(C^*), \qquad \text{size}_{\{\text{H}\}}(C) = \text{size}_{\{+\}}(C^*),$$
$$\text{size}_{\{*_a\}}(C) = \text{size}_{\{*_a\}}(C^*) \quad \textit{for any } a \in R,$$
$$\text{size}_{\{0\}}(C) = \text{size}_{\{\omega\}}(C^*), \qquad \text{size}_{\{\omega\}}(C) = \text{size}_{\{0\}}(C^*).$$

A circuit is limited to compute one specific function with inputs and outputs of fixed size (in term of elements of R). However complexity theory is interested in algorithms that compute on inputs of variable size. This leads to study families of circuits.

DEFINITION 7 (Circuit family). *Let R be a ring, $\mathcal{B}$ a basis over R and $\mathcal{P}$ a set. A* circuit family *over $(R, \mathcal{B}, \mathcal{P})$ is a family of circuits over $(R, \mathcal{B})$ indexed by $\mathcal{P}$. $\mathcal{P}$ is called the* parameter space *of the family. When the mapping from $\mathcal{P}$ to the circuits is Turing-computable, the family is called* uniform.

We are mainly interested in uniform circuit families since they are equivalent to computable functions, theorem 1 easily generalizes to them. We will not study uniform circuit families more in depth, what we do instead is directly work on computer programs implicitly representing circuit families and automatically deduce the transposed family without actually using the circuit model. More details on uniform circuit families can be found in [24].

3. `transalpyne`

`transalpyne` is a programming language suitable for expressing linear algebraic programs and automatically transpose them. Its compiler/interpreter is able to implicitly deduce which families of circuits a given program is equivalent to and to produce a program computing the transposed family.

3.1 Concepts

`transalpyne` has been conceived as a scripting language to be used on top of computer algebra systems. We made an effort to give syntax and semantics as close as possible to the python programming language.

In `transalpyne` there is no such concept as an executable program: only functions can be defined in `transalpyne`. We call *target language* the language to which `transalpyne` programs are compiled; the output of compilation is a library file whose functions can be imported by programs written in the target language. Only compilation to python is supported for the moment.

`transalpyne` can also be interpreted via the python interpreter. A `transalpyne` library contained in a file `my-library.yp` can be imported in a python program via the statement

```
import my-library
```

The python interpreter recognizes the `.yp` extension and launches the `transalpyne` interpreter on the file; the functions of the library are interpreted and transposed by the `transalpyne` interpreter and their names are exported to the python namespace.

`transalpyne` is mostly dynamically typed, with the only exception of *algebraic types*. In order to transpose a function, `transalpyne` must know at transpose time which variables contain algebraic elements and which variables contain other data (such as booleans, strings, ints, etc.); this can be done by explicitly specifying the type of the input and output parameters of a function, while all the other variables can be left untyped. `transalpyne` supports two sorts of algebraic types: ring elements and module elements; we plan to support more complex algebraic types, such as algebras, in the future. `transalpyne` relies on python's operator overloading to represent ring operations.

Before transposing a function, `transalpyne` must *prove* that it is indeed a linear function in its arguments. The technique it uses is to *linearize* the function, that is to make certain input and output parameters constant until it can be shown that the remaining output parameters are linear in the remaining input parameters. We discuss this in Section 4.

3.2 Syntax

We only describe `transalpyne` syntax informally. Indentation has a syntactic value (it delimits blocks) and keywords are pretty much the same. A `transalpyne` file contains a *type declaration* section followed by a *name definition* section.

3.2.1 Type declarations

`transalpyne` supports two type constructors: a ring constructor and a free module constructor.

```
type Ring R
type Module(R) M
```

This example declares `R` as a ring type and `M` as a free module type over the ring `R`. The typechecker ensures that modules are consistently declared.

3.2.2 Name declarations

Name declarations take three forms: *imports*, *function definitions* and *aliases*. Imports are declared as in python and have the same semantics. Note however that the linearization algorithm considers any imported function as a constant function.

There is no `return` statement in `transalpyne`, function definitions are declared as follows

```
def (a, b)my-function(c, d):
```

where input arguments are given on the right and output arguments on the left.

Inside function definitions, there are four types of statements: `pass` statements[1], assignments (including augmented assignments), `for` loops and `if`s. The syntax is identical to python's.

On the left hand side of assignments, may only appear variable names and subscripts. On the right hand side of assignments, the following types of expressions may appear:

- String, numeric and boolean constants;
- Binary and unary operators `+`, `-`, `*`, `/`, `%`, `div`, `mod`, `<`, `>`, `<=`, `<=`, `==`, `!=`, `and`, `or`, `not`, `in`;
- Parenthesized expressions;
- Subscripts and slices;
- List constructors, including comprehensions;
- Variable evaluations;
- Function calls.

[1] The statement that does nothing.

The syntax for all of these is identical to python's. The only notable exception are function calls where a keyword `trans` is added to let the user call a transposition of a function. In case a function has more than one linearization (and thus more than one transposition), *signature specifiers* enclosed in braces `{`, `}` permit to specify which linearization/transposition is wanted.

Finally, aliases permit to export specific linearization/transpositions of functions with names that can be used inside a python program.

Figure 2 gives a complete `transalpyne` example. It defines a `product` function and two aliases (with transposition and signature specifiers).

```
type Ring R

def (R c)product(R a, R b):
    c = a * b

l_product = trans \
    {linear R}product{linear R, const R}
r_product = trans \
    {linear R}product{const R, linear R}
```

Figure 2. A `transalpyne` program

3.3 Semantics

We only give here the points were `transalpyne` semantics differ from python's.

3.3.1 Types

`transalpyne` is statically typed for algebraic types. The type of each input and output parameter of a function must be specified in the definition as in figure 2. When the type of an argument is omitted, it is assumed to have non-algebraic type. Variables inside the body of a function cannot be explicitly typed, a type-inference algorithm deduces their types from the types of the input parameters.

3.3.2 Side effects

There is no side effect in `transalpyne`. In particular, there is no global variable and assignment itself is more akin to the let-binding of a functional language. After having transposed the functions, the `transalpyne` compiler/interpreter leaves to the target language the task of executing them, thus it cannot enforce the no-side-effect policy at runtime. It is the responsibility of the user to insure that no side effect happens inside a `transalpyne` function.

3.3.3 Algebraic variables

Type declarations merely say that some variables belong to a type, but do not specify any particular implementation of the type. The implementation of rings and modules is left to the user and must be given in an external module written in the target language. The user is only required to implement them as objects and to expose a few methods.

Ring objects must:

- Overload `+` and `*` with the obvious semantic;
- Implement a method `zero` that returns the zero of the ring;
- Optionally, implement a method `one` that returns the one of the ring;
- Optionally, implement a method `Z` that takes an integer n and returns the element $n \cdot 1$ of the ring;
- Optionally, implement methods `div` and `mod` that perform Euclidean division with remainder;
- Optionally, overload `/` thus making the ring into a field.

Module objects must:

- Overload `+` and `*` with the obvious semantic;
- Implement a method `zero` that returns the zero of the module;
- Overload the subscript operator `[]` so that it implements some arbitrary projections on the underlying ring. Most often, a module will be implemented as an array of ring objects and `[i]` will just be projection onto the `i`-th coordinate.
- Overload the assignment to subscript operator in the obvious way.

Algebraic output parameters of a function are implicitly initialized to zero via their `zero` method. This insures that non-assigned algebraic output parameters are always linear in the inputs of the function.

Algebraic elements cannot be combined through the use of lists: lists of algebraic objects are non-algebraic objects and extraction from a list always yields a non-algebraic object.

3.3.4 Function calls

`transalpyne` does not have tuples; the return type of a function with many output parameters is not a tuple, as a consequence its return value cannot be assigned to a variable: it must be assigned to as many variables as there are output parameters. Another consequence of this is that functions with many outputs cannot be used inside expressions: their outputs can only be assigned to variables.

Function names not declared in the library are simply regarded as external functions. They are assumed to have one return parameter, thus a multi-assignment will return an error. External functions have no algebraic input or output parameters. This is useful to call builtin python functions[2] from inside a `transalpyne` program.

3.3.5 Recursion and Higher order

`transalpyne` allows recursion and even calling its own transpose. It does not allow to pass functions as arguments to a function, although the transposition algorithm internally uses this technique to transpose for loops. A higher order transposable language is theoretically possible and we plan to add this feature to `transalpyne` in the future.

4. Linearization

The function

```
def (R c)product(R a, R b):
    c = a * b
```

is not linear in `a` and `b`, but it can be made linear by fixing one of the two arguments, for example by considering it as the family of mappings $a \mapsto ab$ for any given b. We call *const* the arguments that are fixed and *linear* the others; clearly const outputs must only depend on const inputs, while linear outputs must linearly depend on linear inputs for any given values of the const inputs. This is equivalent to model the function as a family of circuits whose parameter space is the domain of the const arguments.

`transalpyne` allows the user to annotate the types of the arguments in order to specify whether they are const or linear (non-algebraic arguments are by default const).

[2] One common example is the function `range`, needed to iterate over module elements.

```
def (linear R c)product(linear R a,
                        const R b):
    c = a * b
```

Fortunately, the user need not specify all the modifiers since they can be inferred algorithmically.

We call *signature* a list of linear/const modifiers attached to the arguments of a function. A function can, of course, have more than one signature. The idea behind the signature inference algorithm is simple. `transalpyne` starts from a few axioms on the signature of elementary operators, here is some of them:

```
* : (linear, const) -> linear
    (const, linear) -> linear
    (const, const) -> const
+ : (linear, linear) -> linear
    (const, const) -> const
zero : linear
       const
one : const
```

Then `transalpyne` applies an algorithm similar to the type inference of functional languages [3] to deduce all the possible signatures of a function. If more than one linearization exists, `transalpyne` will generate one transposition for each of them. The user is also allowed to only specify partial information, the compiler/interpreter will restrict to the signatures that match such information or issue an error if no signature matches.

Function calls and aliases use the same principle. The signature specifiers `{...}` let the user specify which of the linearizations of a given function has to be called or saved under a new name. Thus, in the example we gave in figure 2, `l_product` is an alias for the transposed left-linear product, while `r_product` is an alias for the transposed right-linear one. Aliases are extremely useful since they permit to export to the namespace of the target language the transposed functions that could not be accessed otherwise.

5. A word about automatic differentiation

Before discussing the way `transalpyne` transposes programs, we recall some concepts from the theory of Automatic Differentiation (AD).

The transposition principle has often been viewed as a special case of the reverse mode in automatic differentiation [1, 14, 18]. This is somewhat ironic as the whole idea of automatic differentiation can elegantly be derived in the arithmetic circuit model and reverse mode in particular is just an application of the transposition principle [7]. It is probable that the need for efficient AD tools in many scientific areas other than mathematics and computer science is responsible for such reversal of roles.

Here we show how AD can be expressed in the arithmetic circuit model and then discuss the main differences between the AD tools and our approach. A much more complete study on the differentiation of circuits and on how the transposition principle relates the gradient to the differential can be found in [7, 20].

To simplify, we consider a basis $\mathcal{B}$ over $\mathbb{R}$ made exclusively of everywhere continuously derivable functions (w.r.t the standard metric of the Euclidean space $\mathbb{R}^n$). What we give here is a technique to approximate a circuit over $(\mathbb{R}, \mathcal{B})$ by a "linear" circuit.

DEFINITION 8 (Derivative of a circuit). *Let C be a circuit over $(\mathbb{R}, \mathcal{B})$ with n inputs and let $x \in \mathbb{R}^n$. For any function $f \in \mathcal{B}$, we denote by J_f its Jacobian. Then the* derivative *of C at x, denoted by $\mathrm{d}_x C$ is the arithmetic circuit where any $v \in V$ with $\beta(v) = f$ and incident edges $e_1, \ldots, e_m$ has been substituted by a v' with*

$$\beta(v') = J_f\left(\mathrm{eval}_{e_1}(x), \ldots, \mathrm{eval}_{e_m}(x)\right) . \tag{5}$$

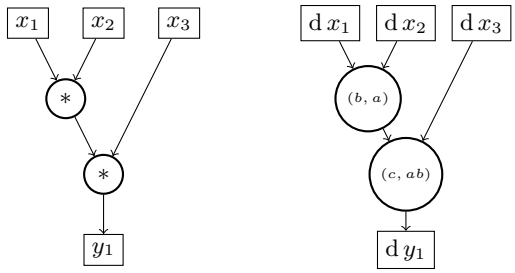

Figure 3. A circuit and its derivative at the point $x = (a, b, c)$. We have replaced multiplication nodes with linear applications represented by 1×2 matrices.

Taking the derivative of a circuit at x amounts to chose for each node a linear approximation at the point where it is evaluated. It is clear that this yields a linear approximation for the circuit at x.

PROPOSITION 1. $\mathrm{eval}_{\mathrm{d}_x C} = J_{\mathrm{eval}_C}(x)$.

It is also clear that $\mathrm{d}_x C$ is defined over a basis that is exclusively made of matrices with coefficients in $\mathbb{R}$. These circuits are slightly more general than those over the basis $\mathcal{L}$, but it is easy to generalize theorem 1 to them. In other words we have defined a transformation from circuits computing derivable functions to linear circuits.

Now $\mathrm{d}_x C$ can be queried by black-box algorithms to obtain information about the Jacobian $J_{\mathrm{eval}_C}(x)$. The simplest application is to compute the directional derivative in x along a direction u: for this task it suffices to evaluate the circuit once, since $\mathrm{eval}_{\mathrm{d}_x C}(u)$ is the desired value. Computing the derivative along n linearly independent directions yields the whole Jacobian and this roughly corresponds to the direct mode in automatic differentiation[3].

When the circuit has many inputs but only one output, there is a more convenient way to get the whole gradient with only one black-box query: $\mathrm{d}_x C$ computes a linear form whose coefficients are exactly the coefficients of the gradient, thus the dual circuit $(\mathrm{d}_x C)^*$ computes the transposed form, or column vector. The single query $\mathrm{eval}_{(\mathrm{d}_x C)^*}(1)$ yields this vector. This is exactly what is called "reverse mode" in automatic differentiation.

Note however that one is not limited to direct or reverse mode: any black-box algorithm can be combined with the derivative circuit to obtain information on the original function. For example Wiedemann's algorithm [25] can be used to determine if the function is invertible around x, and the directional derivatives of the inverse can be computed.

Of course, direct and reverse automatic differentiation can be defined by the more classical chain rule, and then the transposition theorem can be derived as a special case of the reverse mode by observing that, when all the nodes of the circuit are linear maps, $C = \mathrm{d}_x C$ for any x. After all, the code transformation techniques given in [1] and developed in the next section were already invented by researchers in AD [10], though not often implemented.

So, why invent `transalpyne` when there is already plenty of AD tools out there? The answer is manifold and we only list here some key points.

- AD is often interested in recovering the full Jacobian, instead of just having a black-box for it. For an $n \times m$ matrix, this requires n queries in direct mode or m queries in reverse mode. In both cases, AD tools do more work than what we would like to.

[3] To be more precise, direct mode automatic differentiation constructs $\mathrm{d}_x C$ and evaluates the n directions in parallel, thus reducing the amount of storage needed.

- Many AD tools do not optimize the computation of $\mathrm{d}_x C$ for the case where nodes are linear and still compute the whole circuit. In particular, many AD tools generate a graph representation of an arithmetic circuit from a program instead of directly transposing the code. This adds a constant overhead to the case of transposition where simply $\mathrm{d}_x C = C$.
- If the circuit $\mathrm{d}_x C$ is computed, it must be fully stored in memory for reverse mode. This may seem innocuous as $\mathrm{d}_x C$ has the same size as C, but consider programs that compute eval_C by means of for loops or other iterative constructs: while the evaluation of C is compact and cheap, the evaluation of $\mathrm{d}_x C$ possibly requires to introduce a new variable for each iteration of the loop. Depending on the implementation, this may lead to code or storage bloat. In the case of transposition, this never happens since for loops are directly reversed (at least when all the variables are linear). Griewank [11] gives a time/memory compromise that permits to keep both storage and time in a factor of $\log n$ from the original program, but this is still not as good as transposition.
- Our approach is more general in that it permits to automatically treat functions that depend both on linear and non-linear arguments without any help from the user. Thanks to this, we are able to treat recursive functions, while only few AD tools can.
- Our approach is algebraic and permits to prove bounds on the algebraic complexity of the generated programs, while AD tools usually only deal with floating point numbers. More generally, AD languages are usually less rich than `transalpyne`.

6. Transpostion

After the linearization phase, `transalpyne` generates the transposed functions. Linear in- and out- arguments are swapped, while const arguments do not move. The body itself of the function is transformed: formally, it is translated to a family of arithmetic circuits, the circuits are reversed and the result is translated back to a program; in practice we never compute the circuit representation and work directly on the source code.

The key ideas relevant to the transposition of linear programs are in [2, Chap. 13] and [1], but they have their roots in the method of the *adjoint code* for automatic differentiation, a survey can be found in [10]. We first discuss transposition of functions with no const arguments, then go to the general case.

Consider the program in figure 4 and assume that `h` and `g` have an unique signature where any argument is linear. The transposition is obtained by reversing the flow and transposing the code line by line. When transposing function calls, one simply replaces the function by its transpose. Also, following definition 6, additions become duplications of variables and double uses of variables (such as `c`) become additions. It is interesting to notice that optimizing the transposed program by sharing the double assignment to `trans h(a)` and transposing again yields an equivalent improvement to the original program.

Handling const variables permits to treat `if` statements and products. In `if` statements each branch is transposed as above; this ultimately permits to treat recursive functions: in fact they are treated no differently than normal functions, as in figure 5.

Observe however that this reversal of code may lead to the situation where a function needs a const argument that has not been computed yet. In automatic differentiation, the same problem appears when applying the reverse mode: in this case a *forward sweep* is needed to precompute the jacobians of all the functions at the point of differentiation, then a *reverse sweep* runs through the code in reverse order applying the transpose of the Jacobian to the input vector; see [10, 11]. In our case all the function calls are linear, thus we do not need to compute the jacobians; but we apply the same technique to predict the values of const variables.

```
def (R a)f(R b, R c):
  x, y = g(b, c)
  a = h(x) + h(c) + y
```

```
def (R b, R c)fT(R a):
  y, x, c = a, trans h(a), trans h(a)
  b, tmp = trans g(x, y)
  c += tmp
```

Figure 4. A program with no const variables and its transpose.

```
def (M a)f(linear M b, n):
  if n > 0:
    a = f(b, n - 1)
    a[n] += R.Z(n) * b[n]
```

```
def (linear M b)fT(M a, n):
  if n > 0:
    b[n] += R.Z(n) * a[n]
    b += trans f(a, n - 1)
```

Figure 5. A recursive program and its transpose.

A pathological example is shown in figure 6. Here `y` is a const variable and its value is needed in order to compute `x` in the reverse sweep; but the value is only computed by a call to `f` or its transpose, thus it can only be known too late in the reverse sweep. The forward sweep permits to compute the value of `y` before it is needed.

```
def (R a, R b)f(R c, R d):
  if d > R.zero():
    x, y = f(c, d - R.one())
    a, b = x * y, y + R.one()
  else:
    a, b = c, d
```

```
def (R c, R b)fT(R a, R d):
  # Forward sweep
  if (d > R.zero()):
    _, y = f(a, d - R.one())
    b = y + R.one()
  else:
    b = d
  # Reverse sweep
  if (d > R.zero()):
    x = a * y
    c, y = trans f(x, d - R.one())
  else:
    c = a
```

Figure 6. A pathological example and its transpose (relative to the signature `{linear R, const R}f{linear R, const R}`) using a forward sweep.

Notice however that combining forward sweeps and recursion has a disruptive effect: the transposed algorithm contains now two

recursive calls and its complexity is much worse than the original algorithm. The solution is to compute in the forward sweep only the values that are needed; and to compute them only once through a lazy approach. In practice, based on the fact that const outputs only depend on const inputs, `transalpyne` generates a *constification* for each signature of each function by stripping out all the linear variables and by replacing function calls with constified function calls. Each time a constified function is evaluated, its output is stored in a memoization table and any future call on the same input values will use the values stored in the table. This technique permits to guarantee that the time complexity of the transposed function obeys the transposition theorem, but the space complexity is potentially increased to be as large as time complexity. This is analogous to what happens in the reverse mode of automatic differentiation; also notice that a technique similar to [11] could be applied in order to obtain a tradeoff between the increases in time and space complexities.

In practice, well written programs will contain few assignments to const variables and pathological functions, such as the one of figure 6, will be rare. For this reason `transalpyne` only implements the lazy approach in the interpreter, while the compiler produces classical code with a forward and a reverse sweep. For the same reason, we did not implement the technique of [11].

Finally, `transalpyne` handles `for` loops by translating them into a tail-recursive function and then transposing it. The resulting transposed function is head-recursive and can be transformed back to a `for` loop, unless it contains a forward sweep, which happens whenever the loop contains assignments of const variables. This is more powerful than the setting of [1] where `for` loops can always be transposed to `for` loops.

7. Conclusion

We presented `transalpyne` a scripting language that is well suited to implement linear algebra algorithms. `transalpyne` has no execution capabilities, but permits to define libraries of (multi-)linear transformations that can be used by a target language, this makes it very useful as a scripting language on top of computer algebra systems. `transalpyne` can be easily integrated in any python-based system: all the user has to do is to make sure its ring elements implement the interface given in Section 3.3.3, then any function written in `transalpyne` is transposed on the fly by the interpreter and can be called by other functions written in python. As more output languages will be added to `transalpyne`'s compiler, integration will be possible with other computer algebra systems or generic user written code, although this requires some more effort by the user.

The main features of `transalpyne` are its ability to discover *linearizations* of computer programs and to *transpose* linear programs. The result of the transposition is almost as time and space efficient as the original program, this permits to quickly and automatically implement pairs of algorithms related by duality that are found in the literature and that required a lot of hard man-work to be derived. Some examples we have in mind are the *power projection* and the *middle product* that so often recur in algebraic algorithms [12, 19, 22]. Having ourselves spent a few weeks deriving and implementing transposed algorithms for [4], we can testify on how useful `transalpyne` would have been at that time!

Hence, we believe that `transalpyne` will prove itself as a useful tool to any computer algebraist. `transalpyne` is open source software released under the CeCILL licencse[4]. We are planning to release the first stable version in the next few months, it will be available at `http://transalpyne.gforge.inria.fr/`.

[4] `http://www.cecill.info/`

References

[1] A. Bostan, G. Lecerf, and É. Schost. Tellegen's principle into practice. In *ISSAC'03*, pages 37–44. ACM, 2003.

[2] P. Bürgisser, M. Clausen, and A. Shokrollahi. *Algebraic complexity theory*. Springer–Verlag, 1997.

[3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. POPL 82*, 207–212, 1982.

[4] L. De Feo and É. Schost Fast Arithmetics in Artin Schreier Towers. In *ISSAC'09*, ACM, 2009.

[5] J.-G. Dumas and C. Pernet and B. D. Saunders On finding multiplicities of characteristic polynomial factors of black-box matrices. In *ISSAC'09*, ACM, 2009.

[6] W. Eberly. Black box frobenius decomposition over small fields. In *ISSAC'00*, ACM, 2000.

[7] S. B. Gashkov, and I. B. Gashkov. On the Complexity of Calculation of Differentials and Gradients. *Discrete Math. Appl.* 15(4), pages 327–350, 2005.

[8] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[9] J. von zur Gathen and V. Shoup. Computing Frobenius maps and factoring polynomials *Comput. Complexity*, vol. 2, 187–224, 1992.

[10] J.-C. Gilbert and G. Le Vey and J. Masse. *La différentiation automatique de fonctions représentées par des programmes*. RR INRIA 1557, 1991.

[11] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Opt. Meth. and Soft.*, 35–54, 1992.

[12] G. Hanrot, M. Quercia, P. Zimmerman. The Middle Product Algorithm I *Applicable Algebra in Engineering, Communication and Computing* vol. 14, 6, pp. 415–438, 2004.

[13] J. Hopcroft and J. Musinski. Duality applied to the complexity of matrix multiplication and other bilinear forms. *SIAM Journal on Computing*, vol. 2, pp. 159–173, 1973.

[14] E. Kaltofen. Challenges of symbolic computation: my favorite open problems. *J. Symb. Comp.*, 29(6):891–919, 2000.

[15] E. Kaltofen and V. Shoup. Fast polynomial factorization over high algebraic extensions of finite fields. In *ISSAC '97*, 184–188. ACM, 1997.

[16] E. Kaltofen and V. Shoup. Subquadratic-time factoring of polynomials over finite fields. *Math. Comput.*, 1179–1197, AMS, 1998.

[17] E. Kaltofen and B. D. Saunders. On Wiedemann's Method of solving sparse linear systems. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, LNCS 539, 29–38, Springer, 1991

[18] E. Kaltofen and L. Yagati. Improved sparse multivariate polynomial interpolation algorithms. In *ISSAC '88*, 467–474, ACM, 1988.

[19] C. Pascal and É. Schost. Change of order for bivariate triangular sets. In *ISSAC'06*, pages 277–284. ACM, 2006.

[20] I. S. Sergeev. On the Complexity of the Gradient of a Rational Function. *J. Applied and Industrial Mathematics*, Vol. 2, No. 3, pages 385–396, 2008.

[21] V. Shoup. Fast construction of irreducible polynomials over finite fields. *J. Symb. Comp.* 17:371-391, 1994.

[22] V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symb. Comp.*, 20(4):363–397, 1995.

[23] G. Villard. Computing the Frobenius normal form of a sparse matrix. In *CASC'00*, pages 395–407, 2000.

[24] H. Vollmer. *Introduction to Circuit Complexity*. Springer-Verlag, 1998.

[25] D. H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Trans. Inf. Theory*, vol. IT-32:54–62, 1986.